Outline
- Theory: Representation of Integers
- Application: C Integers and Size
- Application: Bitwise operators
- Exercise: Activity 6.1

Representation of Integers
- Computers only understand 1 and 0's. You may have heard of this.
- A value that can only be either 0 or 1 is called a bit.
- Everything in a computer's memory is a collection of bits.
- How we interpret those bits is up to us.
- We can interpret those bits to mean integers.

Number Representation
- Normally, we count in Base-10
- Example:
  - 304 = 4 * 100 + 0 * 10 + 3 * 10
  - 4050 = 0 * 1000 + 5 * 100 + 0 * 100 + 4 * 10
- This is our familiar concept of the one's place, the ten's place.
- We can do the same thing with any number
- Typically in computer programming, we deal with numbers in Base 8, 16, 10, and 2.

Binary
- Use powers of two instead of 10.
- \( xyz = x \times 4 + y \times 2 + z \times 1 \)
- \( 101 = 1 \times 4 + 0 \times 2 + 1 \times 1 = 5 \)
- Counting in binary:
  - 0, 1, 10, 11, 100, 101, 110, 111, 1000 ...
- Each place, from right to left is: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 ...
- Try converting the following numbers:
  - 33, 43, 203
  - 10011, 110101.
**Binary Answers**

- Don't Peek!
- 33 -> 100001
- 43 -> 101011
- 203 -> 11001011
- 10011 -> 19
- 110101 -> 53

**C Integers**

- Unsigned integers in C are stored in Binary Coded Decimal (BCD); this is what we were just practicing.
- Char: 1 Byte (8 bits)
- Short: 2 Bytes
- Int: 1 Word (2 - 4 Bytes)
- Long: 4 Bytes
- Signed values are stored in something called 2's Complement; that's beyond the scope of this course

**Bitwise Operations**

- There exist operators for manipulating the bits of values; these are called bitwise operators
- ~ = Negation NOT
- | = Bitwise OR
- & = Bitwise AND
- ^ = Bitwise XOR
- << = Left Shift
- >> = Right Shift

~

- Takes each bit and flips it
- Example:
  - ~33 = ~100001 = 011110
  - ~43 = ~101011 = 010100

| |

- Bit by bit, or's the two numbers
- Example
  - 100001 | 101011 = 101011
  - 111000 | 000111 = 111111

Kenneth W. Flynn (7-12)
12/01/98
&
- Bit by bit and's the two numbers
- Example
  - 100001 & 101011 = 100001
  - 111000 & 000111 = 000000

>>
- Moves the bits right, fills in with a zero on the left
- Example
  - 111 >> 2 = 001
  - 111 >> 1 = 011
  - 101 >> 1 = 010

<<
- Left shift, moves the bits left, filling in with 0's
- Examples:
  - 111 << 1 = 110
  - 111 << 2 = 100
  - 101 << 1 = 010

Ok, But Why?
- Frequently in hardware, we need these kind of manipulations
- Sometimes we want to store 8 Boolean values in a compact way. Using chars to store would mean 8 bytes. One bit can be used with bitwise comparision

Masks
- Frequently, we're only interested in a particular bit within a value (the smallest chunk of bits we can get is 8).
- We *mask* off the other bits so we have a normal true or false value
- Say we're interested in the high order bit...
  - 10011111
  - 10000011
- How can we extract just this bit?

Masks II
- By ANDing the value with 00000001
- This will turn off the first 7 bits, always. The final bit will be on if it was on in the value we're examining, and off if it was off
- This suppresses information about the other bits and always produces either:
  - 00000001 -> 1 -> True
  - 00000000 -> 0 -> False
More Masks

△ 00100000 & x
  □ 00100000 -> 32 -> True
  □ 00000000 -> 0 -> False

The Bits

△ Write a function that takes an unsigned integer and prints out the number in bits. You might want to use a mask and the shifting operators
△ Use this function, read two numbers, display them in binary, negate them, or them, and and them. Print each of these results in binary.